下载APP

09 | RDD常用算子（三）：数据的准备、重分布与持久化

2021-09-29 吴磊

《零基础入门Spark》 课程介绍 >

讲述：吴磊
时长 23:24 大小 21.44M

你好，我是吴磊。

在 RDD 常用算子的前两讲中，我们分别介绍了用于 RDD 内部转换与聚合的诸多算子，今天这一讲，我们继续来介绍表格中剩余部分的算子。

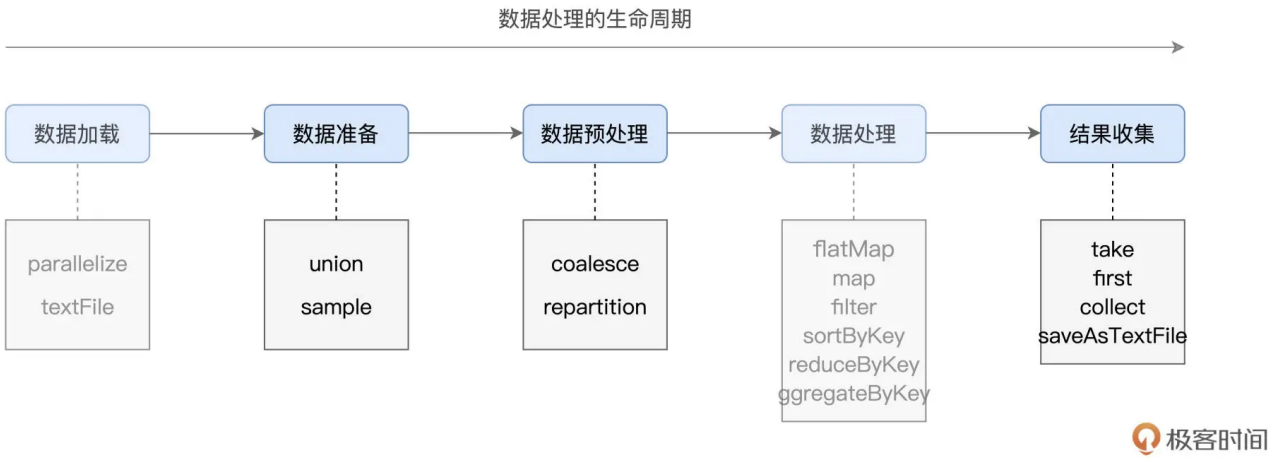
按照惯例，表格中的算子我们不会全都介绍，而是只挑选其中最常用、最具代表性的进行讲解。今天要讲的算子，我用加粗字体进行了高亮显示，你不妨先扫一眼，做到心中有数。

算子类型	适用范围	算子用途	算子集合
Transformations	任意RDD	RDD内数据转换	map mapPartitions mapPartitionsWithIndex flatMap
	Paired RDD	RDD内数据聚合	groupByKey sortByKey reduceByKey aggregateByKey
	任意RDD	RDD间数据整合	union intersection join
	任意RDD	数据整理	sample distinct
	任意RDD	数据分布	coalesce repartition
Actions	任意RDD	数据收集	collect first take takeSample
	任意RDD	数据持久化	saveAsTextFile saveAsSequenceFile
	任意RDD	数据遍历	foreach

极客时间

RDD算子分类表

你可能会觉得，这些高亮显示的算子乍一看也没什么关联啊？但如果我们从数据生命周期的角度入手，给它们归归类，很容易就会发现这些算子分别隶属于生命周期的某个阶段。



极客时间

数据生命周期

结合上图，我们分别来看看每个算子所在的生命周期和它们实现的功能。

首先，在数据准备阶段，union 与 sample 用于对不同来源的数据进行合并与拆分。

我们从左往右接着看，接下来是数据预处理环节。较为均衡的数据分布，对后面数据处理阶段提升 CPU 利用率更有帮助，可以整体提升执行效率。那这种均衡要怎么实现呢？没错，这时就要 coalesce 与 repartition 登场了，它们的作用就是重新调整 RDD 数据分布。

在数据处理完毕、计算完成之后，我们自然要对计算结果进行收集。Spark 提供了两类结果收集算子，一类是像 take、first、collect 这样，把结果直接收集到 Driver 端；另一类则是直接将计算结果持久化到（分布式）文件系统，比如咱们本讲会提到的 saveAsTextFile。

好啦，清楚了我们今天要讲哪些算子，以及它们大致的定位与功用之后，接下来，我们就正式来讲讲这些算子的具体用法。

数据准备

首先，我们先来说说数据准备阶段的 union 和 sample。

union

在我们日常的开发中，union 非常常见，它常常用于把两个类型一致、但来源不同的 RDD 进行合并，从而构成一个统一的、更大的分布式数据集。例如，在某个数据分析场景中，一份数据源来自远端数据库，而另一份数据源来自本地文件系统，要将两份数据进行合并，我们就需要用到 union 这个操作。

具体怎么使用呢？我来举个例子。给定两个 RDD：rdd1 和 rdd2，调用 rdd1.union(rdd2) 或是 rdd1 union rdd2，其结果都是两个 RDD 的并集，具体代码如下：

复制代码

```
// T：数据类型
val rdd1: RDD[T] = _
val rdd2: RDD[T] = _
val rdd = rdd1.union(rdd2)
// 或者rdd1 union rdd2
```

需要特别强调的是，union 操作能够成立的前提，就是**参与合并的两个 RDD 的类型必须完全一致**。也就是说，RDD[String]只能与 RDD[String]合并到一起，却无法与除 RDD[String]以外的任何 RDD 类型（如 RDD[Int]、甚至是 RDD[UserDefinedClass]）做合并。

对于多个类型一致的 RDD，我们可以通过连续调用 union 把所有数据集合并在一起。例如，给定类型一致的 3 个 RDD：rdd1、rdd2 和 rdd3，我们可以使用如下代码把它们合并在一起。

复制代码

```
// T：数据类型
val rdd1: RDD[T] = _
val rdd2: RDD[T] = _
val rdd3: RDD[T] = _

val rdd = (rdd1.union(rdd2)).union(rdd3)
// 或者 val rdd = rdd1 union rdd2 union rdd3
```

不难发现，union 的典型使用场景，是把多份“小数据”，合并为一份“大数据”，从而充分利用 Spark 分布式引擎的并行计算优势。

与之相反，在一般的数据探索场景中，我们往往只需要对一份数据的子集有基本的了解即可。例如，对于一份体量在 TB 级别的数据集，我们只想随机提取其部分数据，然后计算这部分子集的统计值（均值、方差等）。

那么，面对这类把“大数据”变成 “小数据”的计算需求，Spark 又如何进行支持呢？这就要说到 RDD 的 sample 算子了。

sample

RDD 的 sample 算子用于对 RDD 做随机采样，从而把一个较大的数据集变为一份“小数据”。相较其他算子，sample 的参数比较多，分别是 withReplacement、fraction 和 seed。因此，要在 RDD 之上完成数据采样，你需要使用如下的方式来调用 sample 算子：sample(withReplacement, fraction, seed)。

其中，withReplacement 的类型是 Boolean，它的含义是“采样是否有放回”，如果这个参数的值是 true，那么采样结果中可能会包含重复的数据记录，相反，如果该值为 false，那么采样结果不存在重复记录。

fraction 参数最好理解，它的类型是 Double，值域为 0 到 1，其含义是采样比例，也就是结果集与原数据集的尺寸比例。seed 参数是可选的，它的类型是 Long，也就是长整型，用于控制每次采样的结果是否一致。光说不练假把式，我们还是结合一些示例，这样才能更好地理解 sample 算子的用法。

复制代码

```
// 生成0到99的整型数组
val arr = (0 until 100).toArray
// 使用parallelize生成RDD
val rdd = sc.parallelize(arr)

// 不带seed，每次采样结果都不同
rdd.sample(false, 0.1).collect
// 结果集：Array(11, 13, 14, 39, 43, 63, 73, 78, 83, 88, 89, 90)
rdd.sample(false, 0.1).collect
// 结果集：Array(6, 9, 10, 11, 17, 36, 44, 53, 73, 74, 79, 97, 99)

// 带seed，每次采样结果都一样
rdd.sample(false, 0.1, 123).collect
// 结果集：Array(3, 11, 26, 59, 82, 89, 96, 99)
rdd.sample(false, 0.1, 123).collect
// 结果集：Array(3, 11, 26, 59, 82, 89, 96, 99)

// 有放回采样，采样结果可能包含重复值
rdd.sample(true, 0.1, 456).collect
// 结果集：Array(7, 11, 11, 23, 26, 26, 33, 41, 57, 74, 96)
rdd.sample(true, 0.1, 456).collect
// 结果集：Array(7, 11, 11, 23, 26, 26, 33, 41, 57, 74, 96)
```

我们的实验分为 3 组，前两组用来对比添加 seed 参数与否的差异，最后一组用于说明 withReplacement 参数的作用。

不难发现，在不带 seed 参数的情况下，每次调用 sample 之后的返回结果都不一样。而当我们使用同样的 seed 调用算子时，不论我们调用 sample 多少次，每次的返回结果都是一致的。另外，仔细观察第 3 组实验，你会发现结果集中有重复的数据记录，这是因为 withReplacement 被置为 true，采样的过程是“有放回的”。

好啦，到目前为止，数据准备阶段常用的两个算子我们就讲完了。有了 union 和 sample，你就可以随意地调整分布式数据集的尺寸，真正做到收放自如。

数据预处理

接下来，在数据预处理阶段，我们再来说说负责数据重分布的两个算子：repartition 和 coalesce。

在了解这两个算子之前，你需要先理解并行度这个概念。所谓并行度，它实际上就是 RDD 的数据分区数量。还记得吗？RDD 的 partitions 属性，记录正是 RDD 的所有数据分区。因此，RDD 的并行度与其 partitions 属性相一致。

开发者可以使用 repartition 算子随意调整（提升或降低）RDD 的并行度，而 coalesce 算子则只能用于降低 RDD 并行度。显然，在数据分布的调整方面，repartition 灵活度更高、应用场景更多，我们先对它进行介绍，之后再去看看 coalesce 有什么用武之地。

repartition

一旦给定了 RDD，我们就可以通过调用 repartition(n) 来随意调整 RDD 并行度。其中参数 n 的类型是 Int，也就是整型，因此，我们可以把任意整数传递给 repartition。按照惯例，咱们还是结合示例熟悉一下 repartition 的用法。

复制代码

```
// 生成0到99的整型数组
val arr = (0 until 100).toArray
// 使用parallelize生成RDD
val rdd = sc.parallelize(arr)

rdd.partitions.length
// 4

val rdd1 = rdd.repartition(2)
rdd1.partitions.length
// 2

val rdd2 = rdd.repartition(8)
rdd2.partitions.length
// 8
```

首先，我们通过数组创建用于实验的 RDD，从这段代码里可以看到，该 RDD 的默认并行度是 4。在我们分别用 2 和 8 来调整 RDD 的并行度之后，通过计算 RDD partitions 属性的长度，我们发现新 RDD 的并行度分别被相应地调整为 2 和 8。

看到这里，你可能还有疑问：“我们为什么需要调整 RDD 的并行度呢？2 和 8 看上去也没什么实质性的区别呀”。

在 RDD 那一讲（第 2 讲），我们介绍过，每个 RDD 的数据分区，都对应着一个分布式 Task，而每个 Task 都需要一个 CPU 线程去执行。

因此，RDD 的并行度，很大程度上决定了分布式系统中 CPU 的使用效率，进而还会影响分布式系统并行计算的执行效率。并行度过高或是过低，都会降低 CPU 利用率，从而白白浪费掉宝贵的分布式计算资源，因此，合理有效地设置 RDD 并行度，至关重要。

这时你可能会追问：“既然如此，那么我该如何合理地设置 RDD 的并行度呢？”坦白地说，这个问题并没有固定的答案，它取决于系统可用资源、分布式数据集大小，甚至还与执行内存有关。

不过，结合经验来说，**把并行度设置为可用 CPU 的 2 到 3 倍，往往是个不错的开始**。例如，可分配给 Spark 作业的 Executors 个数为 N，每个 Executors 配置的 CPU 个数为 C，那么推荐设置的并行度坐落在 $N*C*2$ 到 $N*C*3$ 这个范围之间。

尽管 repartition 非常灵活，你可以用它随意地调整 RDD 并行度，但是你也需要注意，这个算子有个致命的弊端，那就是它会引入 Shuffle。

我们知道（第 6 讲详细讲过），由于 Shuffle 在计算的过程中，会消耗所有类型的硬件资源，尤其是其中的磁盘 I/O 与网络 I/O，因此 Shuffle 往往是作业执行效率的瓶颈。正是出于这个原因，在做应用开发的时候，我们应当极力避免 Shuffle 的引入。

但你可能会说：“如果数据重分布是刚需，而 repartition 又必定会引入 Shuffle，我该怎么办呢？”如果你想增加并行度，那我们还真的只能仰仗 repartition，Shuffle 的问题自然也就无法避免。但假设你的需求是降低并行度，这个时候，我们就可以把目光投向 repartition 的孪生兄弟：coalesce。

coalesce

在用法上，coalesce 与 repartition 一样，它也是通过指定一个 Int 类型的形参，完成对 RDD 并行度的调整，即 coalesce (n)。那两者的用法到底有什么差别呢？我们不妨结合刚刚的代码示例，来对比 coalesce 与 repartition。

复制代码

```
// 生成0到99的整型数组
val arr = (0 until 100).toArray
// 使用parallelize生成RDD
val rdd = sc.parallelize(arr)

rdd.partitions.length
// 4

val rdd1 = rdd.repartition(2)
rdd1.partitions.length
// 2

val rdd2 = rdd.coalesce(2)
rdd2.partitions.length
// 2
```

可以看到，在用法上，coalesce 与 repartition 可以互换，二者的效果是完全一致的。不过，如果我们去观察二者的 DAG，会发现同样的计算逻辑，却有着迥然不同的执行计划。

```
scala> rdd1.toDebugString
res17: String =
(2) MapPartitionsRDD[22] at repartition at <console>:25 []
 |  CoalescedRDD[21] at repartition at <console>:25 []
 |  ShuffledRDD[20] at repartition at <console>:25 []
 +-(4) MapPartitionsRDD[19] at repartition at <console>:25 []
    |  ParallelCollectionRDD[0] at parallelize at <console>:26 []

scala> rdd2.toDebugString
res18: String =
(2) CoalescedRDD[23] at coalesce at <console>:25 []
 |  ParallelCollectionRDD[0] at parallelize at <console>:26 []
```

coalesce与repartition有着不同的执行计划

在 RDD 之上调用 toDebugString，Spark 可以帮我们打印出当前 RDD 的 DAG。尽管图中的打印文本看上去有些凌乱，但你只要抓住其中的一个关键要点就可以了。

这个关键要点就是，在 toDebugString 的输出文本中，每一个带数字的小括号，比如 rdd1 当中的“(2)”和“(4)”，都代表着一个执行阶段，也就是 DAG 中的 Stage。而且，不同的 Stage 之间，会通过制表符（Tab）缩进进行区分，比如图中的“(4)”显然要比“(2)”缩进了一段距离。

对于 toDebugString 的解读，你只需要掌握到这里就足够了。学习过调度系统之后，我们已经知道，在同一个 DAG 内，不同 Stages 之间的边界是 Shuffle。因此，观察上面的打印文本，我们能够清楚地看到，repartition 会引入 Shuffle，而 coalesce 不会。

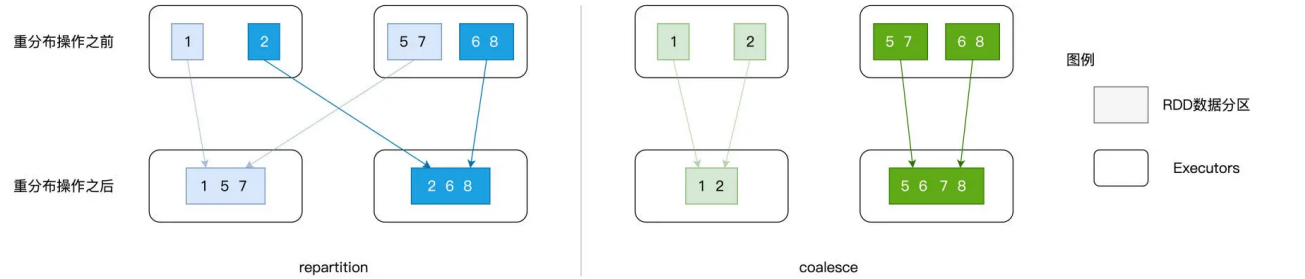
那么问题来了，同样是重分布的操作，为什么 repartition 会引入 Shuffle，而 coalesce 不会呢？原因在于，二者的工作原理有着本质的不同。

给定 RDD，如果用 repartition 来调整其并行度，不论增加还是降低，对于 RDD 中的每一条数据记录，repartition 对它们的影响都是无差别的数据分发。

具体来说，给定任意一条数据记录，repartition 的计算过程都是先哈希、再取模，得到的结果便是该条数据的目标分区索引。对于绝大多数的数据记录，目标分区往往坐落在另一个 Executor、甚至是另一个节点之上，因此 Shuffle 自然也就不可避免。

coalesce 则不然，在降低并行度的计算中，它采取的思路是把同一个 Executor 内的不同数据分区进行合并，如此一来，数据并不需要跨 Executors、跨节点进行分发，因而自然不会引入 Shuffle。

这里我还特意准备了一张示意图，更直观地为你展示 repartition 与 coalesce 的计算过程，图片文字双管齐下，相信你一定能够更加深入地理解 repartition 与 coalesce 之间的区别与联系。



repartition与coalesce计算原理示意图

好啦，到此为止，在数据预处理阶段，用于对 RDD 做重分布的两个算子我们就讲完了。掌握了 repartition 和 coalesce 这两个算子，结合数据集大小与集群可用资源，你就可以随意地对 RDD 的并行度进行调整，进而提升 CPU 利用率与作业的执行性能。

结果收集

预处理完成之后，数据生命周期的下一个阶段是数据处理，在这个环节，你可以使用 RDD 常用算子（二）那一讲介绍的各类算子，去对数据进行各式各样的处理，比如数据转换、数据过滤、数据聚合，等等。完成处理之后，我们自然要收集计算结果。

在结果收集方面，Spark 也为我们准备了丰富的算子。按照收集路径区分，这些算子主要分为两类：第一类是把计算结果从各个 Executors 收集到 Driver 端，第二个类是把计算结果通过 Executors 直接持久化到文件系统。在大数据处理领域，文件系统往往指的是像 HDFS 或是 S3 这样的分布式文件系统。

first、take 和 collect

我们今天要介绍的第一类算子有 first、take 和 collect，它们的用法非常简单，按照老规矩，我们还是使用代码示例进行讲解。这里我们结合第 1 讲的 Word Count，分别使用 first、take 和 collect 这三个算子对不同阶段的 RDD 进行数据探索。

复制代码

```
import org.apache.spark.rdd.RDD
val rootPath: String = _
val file: String = s"${rootPath}/wikiOfSpark.txt"
// 读取文件内容
val lineRDD: RDD[String] = spark.sparkContext.textFile(file)

lineRDD.first
// res1: String = Apache Spark

// 以行为单位做分词
val wordRDD: RDD[String] = lineRDD.flatMap(line => line.split(" "))
val cleanWordRDD: RDD[String] = wordRDD.filter(word => !word.equals(""))

cleanWordRDD.take(3)
// res2: Array[String] = Array(Apache, Spark, From)
// 把RDD元素转换为（Key，Value）的形式
val kvRDD: RDD[(String, Int)] = cleanWordRDD.map(word => (word, 1))
// 按照单词做分组计数
val wordCounts: RDD[(String, Int)] = kvRDD.reduceByKey((x, y) => x + y)

```

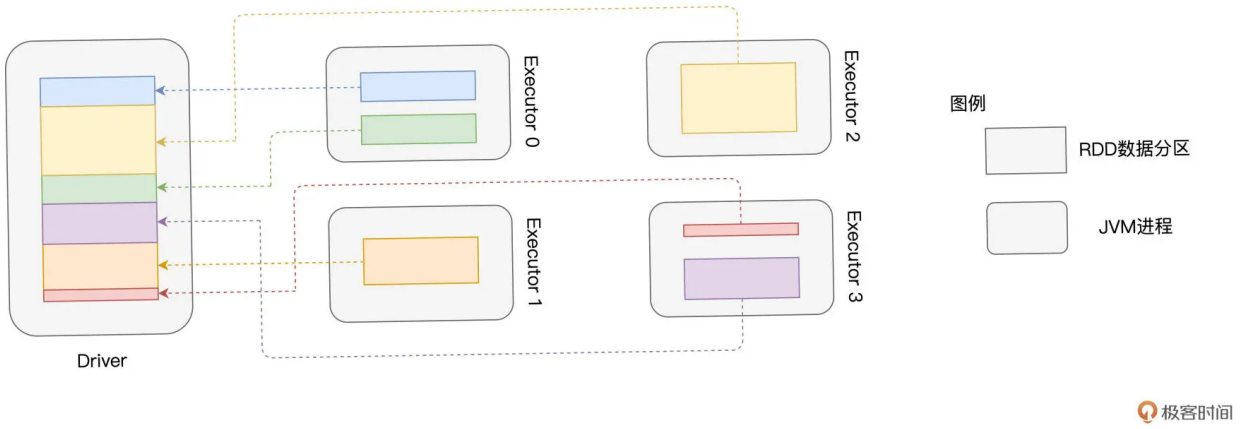
```
21 wordCounts.collect
22 // res3: Array[(String, Int)] = Array((Because,1), (Open,1), (impl...
```

其中，first 用于收集 RDD 数据集中的任意一条数据记录，而 take(n: Int) 则用于收集多条记录，记录的数量由 Int 类型的参数 n 来指定。

不难发现，first 与 take 的主要作用，在于数据探索。对于 RDD 的每一步转换，比如 Word Count 中从文本行到单词、从单词到 KV 转换，我们都可以用 first 或是 take 来获取几条计算结果，从而确保转换逻辑与预期一致。

相比之下，collect 拿到的不是部分结果，而是全量数据，也就是把 RDD 的计算结果全量地收集到 Driver 端。在上面 Word Count 的例子中，我们可以看到，由于全量结果较大，屏幕打印只好做截断处理。

为了让你更深入地理解 collect 算子的工作原理，我把它的计算过程画在了后面的示意图中。



collect算子工作原理

结合示意图，不难发现，collect 算子有两处性能隐患，**一个是拉取数据过程中引入的网络开销，另一个 Driver 的 OOM（内存溢出，Out of Memory）。**

网络开销很好理解，既然数据的拉取和搬运是跨进程、跨节点的，那么和 Shuffle 类似，这个过程必然会引入网络开销。

再者，通常来说，Driver 端的预设内存往往在 GB 量级，而 RDD 的体量一般都在数十 GB、甚至上百 GB，因此，OOM 的隐患不言而喻。collect 算子尝试把 RDD 全量结果拉

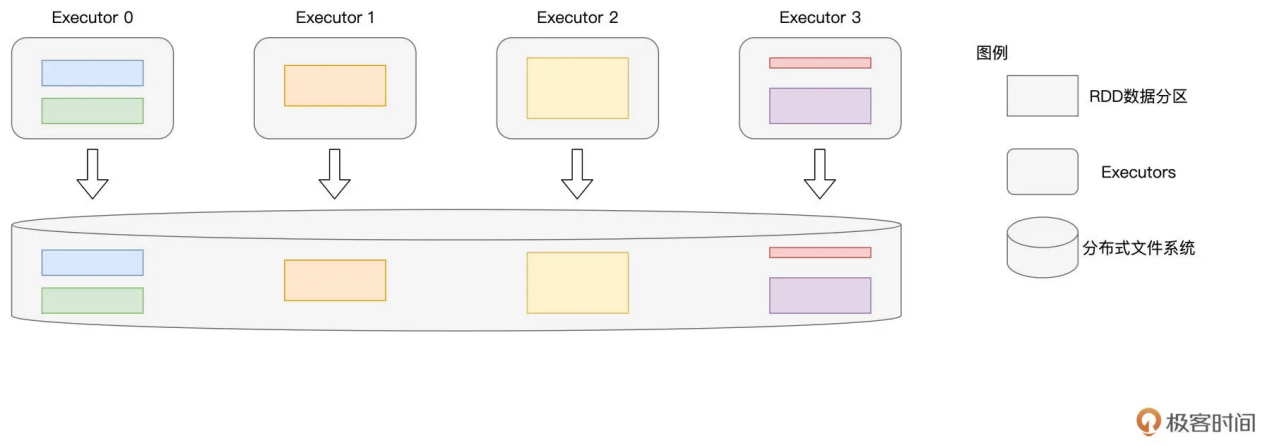
取到 Driver，当结果集尺寸超过 Driver 预设的内存大小时，Spark 自然会报 OOM 的异常（Exception）。

正是出于这些原因，我们在使用 collect 算子之前，务必要慎重。不过，你可能会问：“如果业务逻辑就是需要收集全量结果，而 collect 算子又不好用，那我该怎么办呢？”别着急，我们接着往下看。

saveAsTextFile

对于全量的结果集，我们还可以使用第二类算子把它们直接持久化到磁盘。在这类算子中，最具代表性的非 saveAsTextFile 莫属，它的用法非常简单，给定 RDD，我们直接调用 saveAsTextFile(path: String) 即可。其中 path 代表的是目标文件系统目录，它可以是本地文件系统，也可以是 HDFS、Amazon S3 等分布式文件系统。

为了让你加深对于第二类算子的理解，我把它们的工作原理也整理到了下面的示意图中。可以看到，以 saveAsTextFile 为代表的算子，直接通过 Executors 将 RDD 数据分区物化到文件系统，这个过程并不涉及与 Driver 端的任何交互。



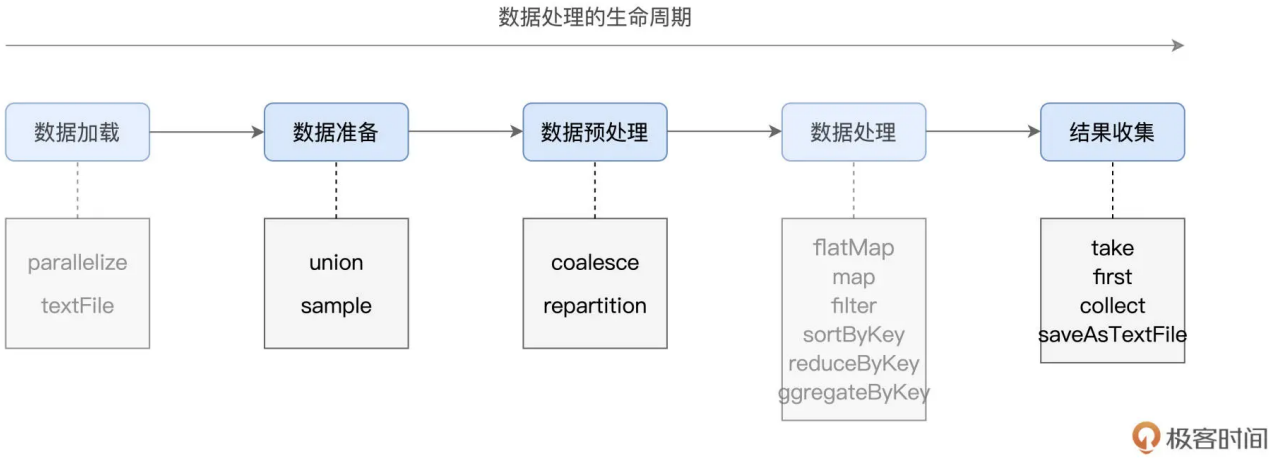
saveAsTextFile工作原理

由于数据的持久化与 Driver 无关，因此这类算子天然地避开了 collect 算子带来的两个性能隐患。

好啦，到此为止，用于结果收集的算子我们就介绍完了，掌握了 first、take、collect 和 saveAsTextFile 等算子之后，你可以先用 first、take 等算子验证计算逻辑的正确性，然后再使用 saveAsTextFile 算子把全量结果持久化到磁盘，以备之后使用。

重点回顾

今天这一讲，我们介绍并讲解了很多 RDD 算子，这些算子可以分别归类到数据生命周期的不同阶段，算子与阶段的对应关系如下图所示。



数据生命周期

在数据准备阶段，你可以使用 union 和 sample 来扩张或是缩小分布式数据集，需要特别注意的是，**参与 union 的多个 RDD 在类型上必须保持一致**。

在数据预处理阶段，你可以利用 repartition 和 coalesce 来调整 RDD 的并行度。RDD 并行度对于 CPU 利用率至关重要，它在很大程度上决定着并行计算的执行效率。一般来说，给定 Executors 个数 N，以及 CPU/Executor 配置个数 C，那么我会**推荐你把 RDD 的并行度设置在 N*C*2 到 N*C*3 之间**。

最后，在结果收集阶段，你可以使用 first、take、collect 等算子来探索数据，这些算子可以用来验证计算过程中的转换逻辑是否与预期一致。当你确认计算逻辑准确无误之后，就可以使用 saveAsTextFile 等算子将全量结果集持久化到（分布式）文件系统。

到今天为止，我们用三讲的篇幅，学习了 RDD 开发 API 中的大部分算子。灵活地运用这些算子，你就能轻松应对日常开发中大部分的业务需求。为了方便你随时回顾、查阅，我把我们一起学过的这些算子整理到了后面的表格中，希望对你有所帮助。

算子类型	适用范围	算子用途	算子	算子作用
Transformations	任意RDD	RDD内数据转换	map	以元素为粒度，使用映射函数f对数据做转换
			mapPartitions	以数据分区为粒度，使用映射函数f对数据做转换
			flatMap	分两阶段转换数据：第一阶段使用映射函数， 把元素转换为集合，第二阶段把集合元素展开，完成计算
			filter	以元素为粒度，使用判定函数过滤数据
	Paired RDD	RDD内数据聚合	groupByKey	以Key为单位，分组收集同一个Key的Value列表
			sortByKey	以Key为单位，对数据集做排序
			reduceByKey	给定reduce函数，以Key为单位，分组聚合同一个Key对应的Value列表
			aggregateByKey	给定初始值、Map阶段聚合函数、Reduce阶段聚合函数，以Key为单位，对数据做分组聚合
	任意RDD	RDD间数据整合	union	对类型完全一致的RDD做合并
	任意RDD	数据整理	sample	给定采样率，对RDD数据集做统计抽样
	任意RDD	数据分布	coalesce	给定目标并行度N，降低RDD并行度
			repartition	给定目标并行度N，重新调整RDD并行度
Actions	任意RDD	数据收集	collect	将RDD数据集全量收集到Driver端
			first	随机提取RDD数据集中的任意一条记录
			take	随机提取RDD数据集中的任意N条记录
	任意RDD	数据持久化	saveAsTextFile	直接通过Executors将RDD数据集物化到文件系统

极客时间

RDD常用算子一览

每课一练

1. 给定 3 个 RDD，除了使用 rdd1 union rdd2 union rdd3 把它们合并在一起之外，你认为还有其他更加优雅的写法吗？（提示：reduce）

2. 相比 repartition，coalesce 有哪些可能的潜在隐患？（提示：数据分布）

欢迎你在留言区跟我交流互动，也推荐你把这一讲分享给更多的同事、朋友，帮他理清 RDD 的常用算子。

分享给需要的人，Ta订阅后你可得 20 元现金奖励

生成海报并分享

赞 2 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 08 | 内存管理：Spark如何使用内存？

下一篇 10 | 广播变量 & 累加器：共享变量是用来做什么的？

更多学习推荐

精选留言 (7)

写留言

AIK 置顶

2021-09-29

#合并RDD
测试了三种方法，分别是union、reduce、++，并且通过调用toDebugString方法查看，显示结果是一致的，下面的代码是在spark-shell上测试的
```scala
scala> val rdd1 = spark.sparkContext.parallelize(1 to 10)...
```
展开

作者回复: 优秀！👍👍👍 满分💯，置顶

Geek_2dfa9a

2021-09-30

第一题
没明白考点是啥，考的是scala的语法么？
val rdd1 = sc.textFile("")
val rdd2 = sc.textFile("")
val rdd3 = sc.textFile("")...

展开

作者回复: 正解，满分💯 👍

1

qinsi

2021-09-29

从文中的描述来看，coalesce似乎并不能避免shuffle？极端的例子，coalesce(1)必然会把数据都放入同一个Executor里？

展开

作者回复: 非常好的问题，我们来细说说~ 取决于你如何调用coalesce(1, shuffle = false/true)，分两种情况。

1. shuffle = false，这个时候，coalesce不会引入Shuffle，但是，所有操作，从一开始，并行度都是1，都在一个executor计算，显然，这个时候，整个作业非常慢，奇慢无比

2. shuffle = true，这个时候，coalesce就会引入shuffle，切割stage。coalesce之前，用源数据DataFrame的并行度，这个时候是多个Executors真正的并行计算；coalesce之后，也就是shuffle之后，并行度下降为1，所有父RDD的分区，全部shuffle到一个executor，交给一个task去计算。

显然，相比前一种，这种实现在执行效率上，更好一些。因此，如果业务应用必须要这么做，推荐这一种实现方法。回答你最初的问题，没错，coalesce(1)必然会把数据都放入同一个Executor里

1

Geek_038655

2021-09-29

collect对于大数据分析结果过大导致的OOM问题，用saveAsTextFile解决是不是过于迁就？

为什么不用foreashPartition?

展开

作者回复: 并无优劣之分~

foreachPartition更多地用来写DB和Kafka；而saveAsTextFile用来写文件系统。数据写入标的不同而已，没有优劣上的差别。

当然，也可以用foreachPartition来写文件系统，但是这个时候，就不如直接调用saveAsTextFile这类算子方便。

1

爱吃猫的鱼

2021-09-29

coalesce 会降低同一个 stage 计算的并行度，导致 cpu 利用率不高，任务执行时间变长。我们目前有一个实现是需要将最终的结果写成单个 avro 文件，前面的转换过程可能是各种各样的，我们在最后阶段加上 repartition(1).write().format('avro').mode('overwrite').save('path')。最近发现有时前面的转换过程中有排序时，使用 repartition(1) 有时写得单文件顺序不对，使用 coalesce(1) 顺序是对的，但 coalesce(1) 有性能问题。目前想到可以 co...

展开

作者回复: 想必老弟对coalesce(1, shuffle = false/true)的用法和区别，已经了如指掌，不再赘述。

遗憾的是，驴和熊猫不可兼得，想要高性能的同时做到保序，确实不容易。shuffle = false/true，前者保序，后者高性能，确实两者很难兼得。

实在对不住，对于两者都需要的场景，我暂时还真想不到特别好的办法，除了你说的"先shuffle -> coalesce(1, shuffle = true) -> 再排序 -> 再收集"之外，我现在还真想不到更好的办法，抱歉

1 1

Unknown element

2021-10-04

老师我看了评论区那个关于coalesce(1，shuffle = false)的问题，您说这个时候coalesce不会引入Shuffle，但是所有操作并行度都是1，都在一个executor计算；这里我不太明白，既然数据是分布在多个节点上，又不能用shuffle，那数据是怎么被汇集到一个节点的？

展开

钱鹏 Allen

2021-10-03

1. 方法一：Seq(rdd1,rdd2).reduce(_ union _) 方法二： rdd1 ++ rdd2 (高赞精简版本)
2.大数据量的情况下，相比 repartition，coalesce没有shuffle，可能会导致数据倾斜，即一个分区上有着大量的数据，而另外一个可能没有多少数据。

展开